Multi-Threaded Basic File Encryptor and Decryptor Program

Steven Stefanovski, Wendy Fogland
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: stefanovski@oakland.edu, wtfogland@oakland.edu

***Abstract*—This project aims to develop a multi-threaded program that can decrypt and encrypt text files using a cipher algorithm. Two parallelization strategies are explored and implemented in the program to provide benchmark testing results compared to a conventional sequential implementation method. It was concluded that one of the parallelization implementations provides higher performance of encrypting/decrypting. The program is written in C++ and utilizes the Intel TBB library.**

I. Introduction

Data security and integrity are important topics as more of our everyday lives are integrated with use of technology. Companies and end-users are becoming more aware of the significance of protecting digital assets as situations like data breaches, data leaks, and identity theft are becoming more common. Therefore, the tangibility and emphasis in protecting a person's data has increased. This paper looks at a basic implementation along with the performance of a program with the ability to encrypt and decrypt text files using parallelism. A sequential method is used to compare the performance of the parallel strategies that would be typically found.

II. Caesar and Vigenère algorithms

It was determined that in order for the program to provide encryption and decryption capabilities in the scope of .txt files that could contain letters, numbers, and special characters that a Caesar and Vigenère cipher algorithm combination would achieve this. It is expected that the .txt files inputted into the program are large, therefore a multi-threaded approach is necessary to achieve efficient processing ability for decrypting and encrypting the files.

A Caesar cipher works by "shifting" letters. Each letter is shifted by a fixed number of positions, which then encrypts the word. The algorithm for the encryption process can be written as the following:

$$E_n(x) = (x + n)\, mod 26$$

This shows the encryption of a letter, *x,* shifted by *n.* The traditional Caesar Cipher assumes the letters A-Z are mapped to numbers 1-26. We will be modifying this slightly so that the letters of the alphabet use their ASCII (American Standard Code for Information Interchange) values, instead. The equation below shows our modified approach.

$$E_n(x) = ((ASCII\ value\ of\ x) + n)\, mod\ 255$$

In contrast, the algorithm for decrypting can be seen as the following. The process is similar to encrypting, but instead is shifting back the letters to their original positions based on the number, *x*, as an input:

$$D_n(x) = ((ASCII\ value\ of\ x) - n)\, mod\ 255$$

A Vigenère cipher utilizes a key to perform different Caesar ciphers on each character of the input text. Each character of the key determines the shift of the Caesar cipher of an individual character of the input text. Letters are mapped to numbers 1-26, so the letter "A" in a key would indicate a shift of 1 for the character. For example, if the input text is "ENCRYPTION" and the key is "ABC", the "E" of the input text would be Caesar Ciphered by 1, the first "N" by 2, and the "C" by 3. Once the key reaches its end, the key is used again from the beginning, until every character of the input text has been ciphered. The equation below more clearly summarizes this concept:

$$E_n(x) = (x_n + K_{n\ mod\ (length\ of\ K)})\, mod\ 26$$

where

x: input text, with letters A-Z mapped to 1-26

K: key text, with letters A-Z mapped to 1-26

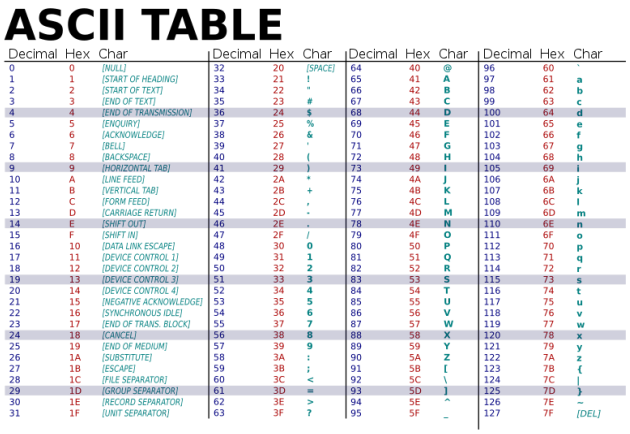
$n = 1$ to length of x

$E_n(x)$: the encrypted value of the input, *x*

We will slightly modify this approach to use ASCII values instead, so:

$$E_n(x) = ((ASCII\ value\ of\ x_n) + (ASCII\ value\ of\ K_n))\, mod\ 255$$

Figure 1 shows a table of the ASCII value for each char (symbol/letter).



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 1: ASCII Table of values for characters

To serve as a proof of concept and to test against the actual software, a sequential MATLAB implementation was developed. Figure 2 shows the output of this MATLAB implementation. The MATLAB script reads in a text file and encodes the text using an ASCII Vigenère Cipher with a key of "chicken". It then takes the encrypted text and decrypts it back to its original form using the same key.

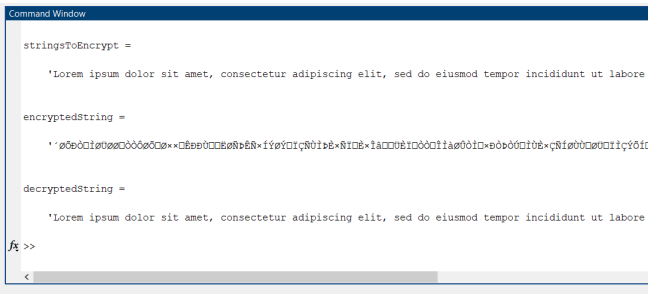


Figure 2: Text from an input file, its encrypted version, and decrypted version

III. Parallelization Strategies

In a sequential execution, it is expected that the time it would take to encrypt a larger text file to be considerable. Therefore, a multi-threaded technique for this style of encryption can optimize the execution time significantly.

Two different parallelization strategies are used and compared: pipelining and parallel_for.

First, a 3-stage pipeline will be implemented to encrypt or decrypt the given text. Similar to Lab 7, stage 1 gets a portion of the input characters as a vector and feeds it into the pipeline. Stage 2 encrypts or decrypts the character array. Stage 3 concatenates the character arrays back into one character array. Figure 3 shows a block diagram of this process.

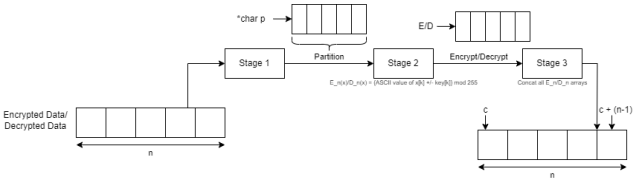


Figure 3: Block Diagram of the pipelining process

The second parallelization strategy utilized was simply using parallel_for in the TBB library. Parallel_for is well suited for this application, since every calculation to encrypt/decrypt a single character of the input text is independent and can be calculated on its own. Since each character can be independently encrypted and the same computation is done to all the characters, parallel_for can easily divide up the iterations to execute parallelly.

IV. Overview of Software

Figure 4 has a basic overview of the program's execution flow. Ease of use for the program was emphasized as the execution was to be run multiple times in a row to get an average of the timing during testing. Some code was reused from previous labs. The functions to read and write the text files were the same in multiple previous labs, and the pipelining portion of the code was modeled after Lab 7.

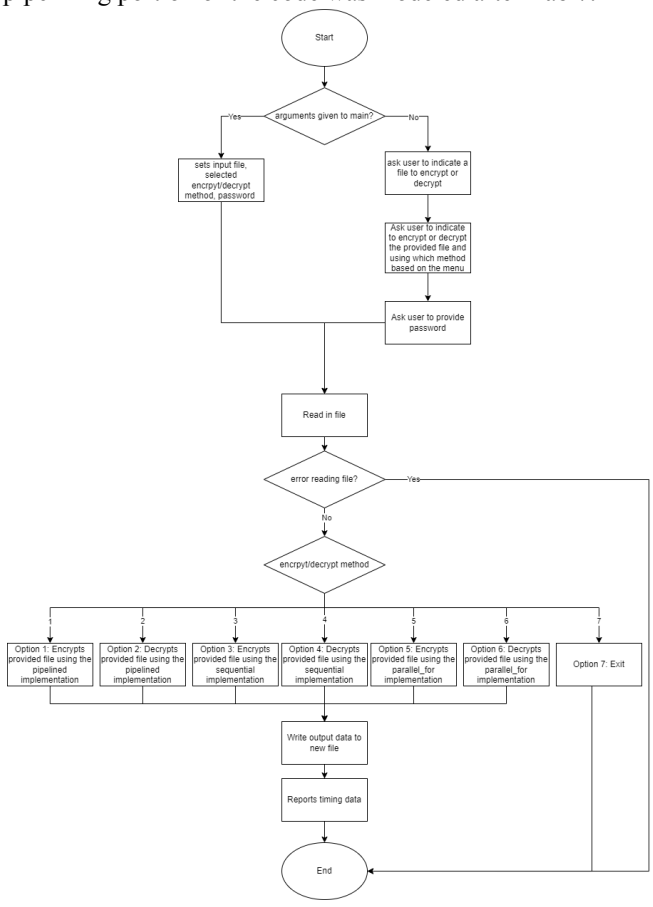


Figure 4: General Flowchart of the encryption/decryption process

V. Software Compiler Setup

For the program that was developed to be executed and tested, a *makefile* was created. This *makefile* configuration was tailored to be used with a g++ compiler as special compiler tags are required. These tags are required in order for the Intel TBB library to be used. The *makefile* contents can be seen in Figure 5.

```
# Compiler/linker setup ------------------------------------------------------

# g++ finalproject.cpp -ltbb -o finalproj -std=c++11

# Linux-specific flags.  Comment these out if using Mac OS X.
PLATFORM = linux
CC       = g++
CFLAGS   = -o -Wall -std=c++11
OSLIBS   =
LDFLAGS  = -ltbb

#-std=c++11: special modifier to allow for lambda expressions in TBB

# make all: all OBJS are compiled and executable available
# make <individual>
# make clean
# '-ltbb' has to be included at the end of the command (can't be in the CFLAGS)
# do not use spaces (tabs are allowed)
# Example programs -----------------------------------------------------------

OBJS = finalproject
all: $(OBJS)

finalproject: finalproject.cpp
    $(CC) $(CFLAGS) finalproject.cpp $(LDFLAGS) -o finalproject

# Library
# lab4_fun.o: lab4_fun.cpp lab4_fun.h
#   $(CC) $(CFLAGS) -c lab4_fun.cpp $(LDFLAGS)

# Maintenance and stuff ------------------------------------------------------
clean:
    rm -f $(OBJS) *.o core
```

Figure 5: Makefile used to compile the software

VI. Performance Benching Setup

An *Intel Atom N2600 Development and Education Board* was utilized to run and benchmark the program developed in this project. The basic specifications of the board can be found below:

- Operating System: Ubuntu 12.04
- 64MB SDRAM x2
- 2MB SSRAM x2
- Dual-Core Processor (Max 1.6GHz)

This hardware was chosen based on what was already being used in class for prior homeworks and labs for timing performance and implementation purposes. Furthermore, the use of the Intel TBB library was necessary for this project and can be used on the board's processor.

To determine the performance of each method of implementation of the cipher algorithm, the "sys/time.h" library was used. Using the "gettimeofday" functionality of the library around just the commutation for each method, the time of execution is calculated in microseconds. Figure 6 shows the use of "gettimeofday" for the sequential method.

```
220         gettimeofday(&start, NULL);
221         cipherTxtFile_seq(fileSize, fileData_In, (char *)
222         gettimeofday(&end, NULL);
```

Figure 6: Formatting of use of "gettimeofday" example in code

In order for the performance of each method to be apparent, different sized input .txt files were utilized. The .txt files were generated using a *Lorem Ipsum* generator which is a standard dummy text typeset to provide normal distribution of letters and characters. This allows for large .txt files sizes where spaces, uppercase letters, lowercase letters, and special characters can be found.

To increase variation of dataset, the following .txt files with a wide range of sizes were determined in Table 1.

File Name	File Size
lorem	4 kB
lorem_long1_7MB	1.7 MB
lorem_long10MB	10 MB

Table 1: Table of file names and their respective sizes

VII. Results

Given the setup described in section VI, the program was executed for each test .txt file five times using encryption and decryption. Figure 7 and Figure 8 show the interface of the program during execution for the different options that are possible with the program. Figure 9 shows a sample of encrypted text, and Figure 10 shows that text from Figure 9 decrypted back to its original form.

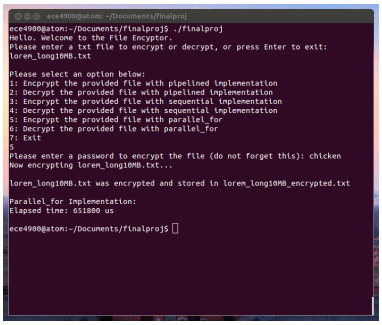


Figure 7: Program interface when encryption is selected for parallel_for method

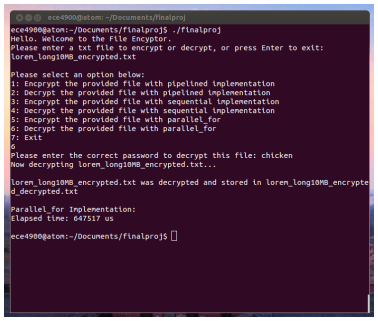


Figure 8: Program interface when decryption is selected for parallel_for method

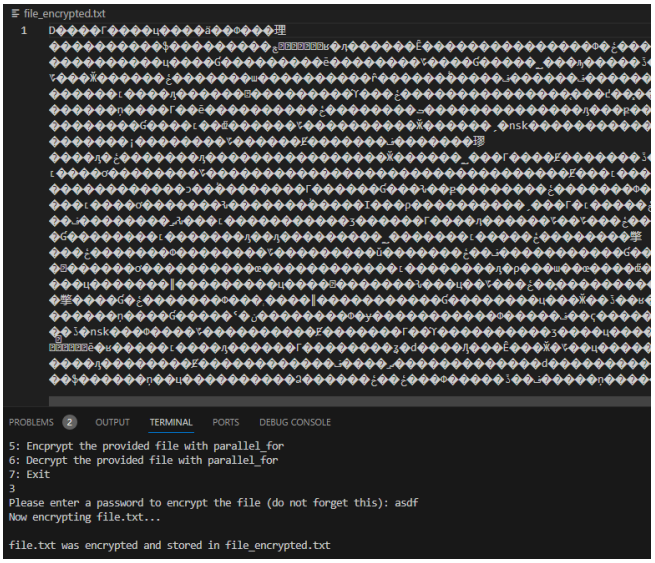


Figure 9: Sample of encrypted text

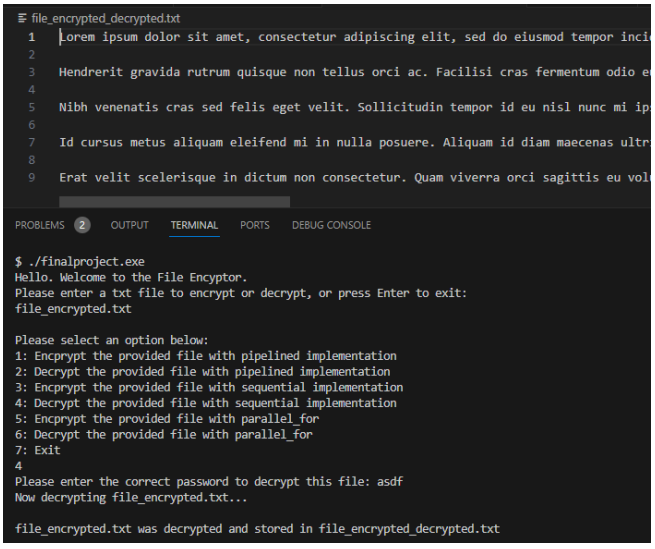


Figure10: The text form Figure 9 decrypted

Table 2 shows the timing results of the three different implementations run with largely different file sizes. As expected, the sequential implementation worked best with smaller file sizes. Once the file sizes started getting above ~1MB, then the parallelization strategies started being beneficial. The parallel_for method was the fastest after this point, providing significant time improvement from the sequential implementation. This supports the prior knowledge of why the method was chosen to be implemented, since parallel_for is optimal for improving the timing of applications where there is a known data size and each element needs the same, but independent, computations performed upon it.

Strangely enough though, the pipeline method was never faster than the sequential implementation. This could be due to the fact that the computations for the Vigenère cipher are not very complex, where there might be more room for the pipeline to improve timing if the computations were more complex. It is also possible that this pipelining method was not very suitable to this application, and more improvements could be made by modifying the pipelining process.

		Timing (us) of different Implementation Method					
		Sequential		Pipeline		Parallel_for	
		Encrypt	Decrypt	Encrypt	Decrypt	Encrypt	Decrypt
File Size	4KB	455	451	12,926	13,984	3,010	2,830
	1.7MB	173,822	166,299	4,544,471	4,547,053	123,601	124,271
	10MB	1,075,845	987,690	27,252,526	27,106,362	658,830	667,193

Figure 10: Timing results for different file sizes & methods

VIII. Conclusions

The program created using a cipher algorithm with different types of execution methods was implemented successfully. It has been shown that given the three methods utilized in the program, the parallel_for implementation had the best performance overall. However, the pipeline implementation performed the worst across the board which supports the idea that parallelizing performance depends on parameters like dataset size, complexity of algorithm, etc. Therefore it is not always the best method performance-wise.

It was also concluded that there is an importance of understanding what types of parallel methods are best suited for in different settings. More exploration could be done in improving performance by expanding on the use of different parallelization methods along with improving data security with trying different ciphering algorithms.

References

[1] https://en.m.wikipedia.org/wiki/File:ASCII-Table-wide.svg
[2] https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
[3] https://moodle.oakland.edu/pluginfile.php/8893205/mod_resource/content/1/Notes%20-%20Unit%204.pdf
[4] https://www.secs.oakland.edu/~llamocca/emb_intel.html
[5] https://www.geeksforgeeks.org/vigenere-cipher/